# 前述

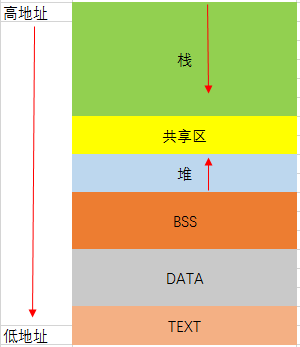
本文档是本人学习做的一些总结，或者帮助自己理解C语言或计算机方面知识的一个记录，

提供给对一些和我一样，对基础上东西感兴趣的人一起探讨，一起学习，文档如果有不正确的地方或者描述不恰当的地方希望大家指正（QQ:1252228983;WX:It-is ok），轻喷！

# 一:基本编程能力

## 变量存储及函数栈帧

一般的，将程序分成几段。如图：



### （1）BSS(Block Standed By Symbol)

指用于存放未初始化或初始化为0的全局变量（包括静态变量，static声明的）一块区域。

### （2）DATA段

指用于存放初始化为非0的全局变量（包括静态变量，static声明的）的一块内存区域。

### （3）RODADA段

存放字符串常量，属于只读区域。

### （4）TEXT段

代码段,指用于存放程序执行代码的一块区域。大小在运行前已经确定，通常为只读。

### （5）堆(heap)

Malloc函数动态分配的内存都属于这个内存区域

### （6）栈（stack）

存放临时变量（局部变量。不包括静态变量），函数返回值等。

### 结论：

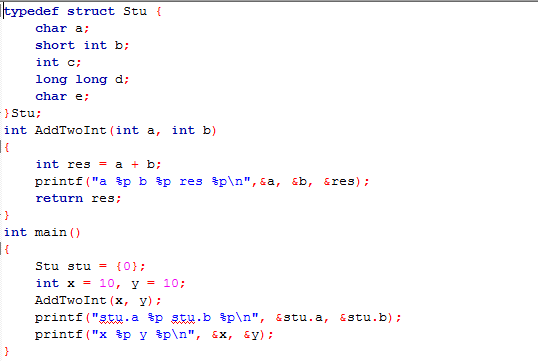
1. 在内存中，地址总是从低到高的，而栈是从高到低的。【堆栈相对增长】

（2）对于一个函数，局部变量（全局变量不会存在栈中）先定义，先分配，因此先声明的变量，其地址总是高。形参也是一种局部变量，

因此右形参变量地址>左右形参变量地址 >内部局部变量地址

（3）结构体的成员变量地址，越靠前，地址越低。

#### 例子1：



针对上述代码：

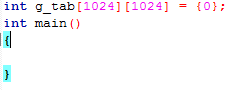
在main函数中： Stu的地址 > x > y

AddTwoInt中： b > a >res

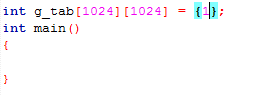
在结构体中：stu.a < stu.b < stu.c

#### 例子2：data内存占文件大小，bss不占

定义方式一：



定义方式二：



方式二中由于数据会存在data段，因此编译出的可执行二进制文件会比方式一的大，并且是4M。



## 2.结构体字节对齐

### (1) 确定结构体的实际对齐长度。

（这个决定后面怎么画内存分布）

a)默认对齐长度：32位默认为4字节，64位默认8字节

b)指定对齐长度#pragma pack(n) n：1/2/4/8/16

c)成员的最大对齐字节

实际对齐长度为MIN(默认，pack, 最大对齐字节长度)的最小值

如果4字节对齐，则内存分布形如：



如果8字节对齐，则内存分布形如：

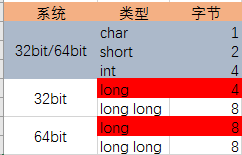


### (2)以画的方式填写内存。

原则如下：

成员存放的地址必须是MIN(结构体实际对齐长度，成员自身的对齐长度)的整数倍。

### (3)友情提醒



### 举例：

Typedef struct Stu {

Char a;

Short b;

Int c;

Long long int d;

Char e;

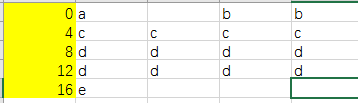
}Stu;

#### 情况一：在32位下，默认4字节对齐，未指定

因为未指定则实际长度取MIN(默认，long long)因此为4字节对齐。

内存分布如下，占用字节数：20字节。

【其中b在第2个字节处，是因为2是MIN(实际对齐长度4，short的对齐长度2)的整数倍】



#### 情况二：在64位下，默认8字节对齐，未指定

未指定，则实际对齐长度取MIN(默认8，long long),为8字节。

内存分布如下，占用字节数：24字节

【其中b,c则在第一行，是因为地址是MIN(实际对齐长度，成员自身对齐长度)的整数倍】



在32位指定为#paragram pack(1) ,1字节对齐时，内存分布如下：

#### 情况三：在32位下,指定为1字节 #pragma pack(1)

指定下，实际长度取MIN(默认的4，指定的1，最大的long long 8)，因此实际对齐长度为1。

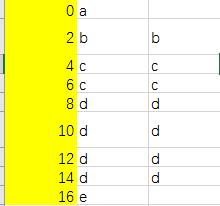
内存分布如下：占用15字节。



【其中b在地址1处，是因为1是MIN(实际对齐长度1，short的2)的整数倍】

#### 情况四：在32位下，指定为2字节 #pragma pack(2)

指定下，结构体的实际对齐长度为MIN(默认的4，指定的1，最大的long long 8)



## 3.结构体与大小端

小端：低字节在低地址，高字节在高地址

大端：低字节在高地址，高字节在低地址

X86 小端

POWER PC 大端

Arm 可配置

由于不论大端小端，栈的地址都是由高到低生长的，并且结构体的成员变量先定义地址越小。

#### 例子1：0x12345678

小端的内存：



更具体化的是，小端的低位在低地址，因此bit位的安排如下：对于字节，低字节在低位，并且低bit也是在低地址



大端的内存:



更具体化的是，大端的低位在高地址，因此bit位的安排如下：对于字节，高字节在低地址，高bit也在低地址

#### 例如2：32位下，求值

typedef struct Stu {

int a:1;

int b:31;

} Stu;

typedef union Uni {

int x;

Stu stu;

}Uni;

Uni uni = {0};

uni.stu.a = 1;

问uni.x = ?

由上面的分析，知道栈的地址都是由低到高。

a）小端的内存其实是



操作uni.stu.a = 1;其实操作了结构体的低地址，因此对应到内存其实低bit位，因此bit0被置1，因此结果为1.

b)大端的内存其实是



操作uni.stu.a = 1;其实操作了结构体的低地址，因此对应到内存其实高bit位，因此bit31被置1，因为是有符号的int，因此是-0,为-2^31。

同理分析如下结构体

Typedef struct Stu {

Int a:1;

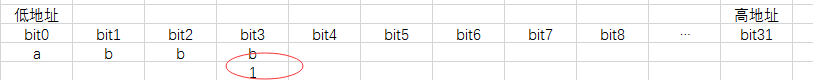
Int b:3

}Stu;

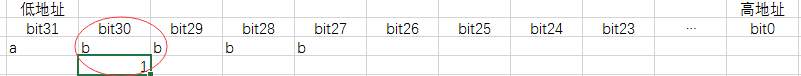
Stu stu = {0};

Stu.b = 4;

小端其实是



大端其实是



## 4.结构体与位域

"位域"是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。

每个域有一个域名，允许在程序中按域名进行操作。

这样就可以把几个不同的对象用一个字节的二进制位域来表示。

如struct BitFiled {

int a:8;

int b:2;

int c:3;

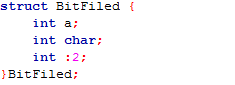
}BitFiled;

内存分布如下：整个结构体占字节为4.（假设32bit下，4字节对齐）



### （1）无名域

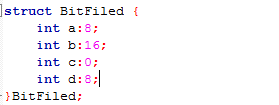
无名域，指的是一个没有指定名称的域，因此不能通过域名访问，但是他占bit因此也是占字节的。



Int:2就是无名域，整个结构体占12字节。（假设32bit下，4字节对齐）

### 占位0bit的域

这种域不占空间，但是要求即使当前空间够的情况下，下一个域名需要重新开一片空间进行存储。



上述定义，c是不占空间的，仅仅告知d需要在另一个int的空间进行定义，因此整个结构体占16字节。（假设32bit下，4字节对齐）

## 5.类型转换

### 5.1源、补码

#### （1）源码：

在最开始的概念的数值的最高位添加符号位，0表示此数值为正数，1表示此数值为负数

#### （2）补码：

正数的源码、反码为自己；负数的补码为对其源码（除符号位外）取反 最后加1.

参与运算的都是补码。只不过正数的补码为自己。

几个易混淆的：

### 5.2符号扩展

有符号数扩展：对现有的已知符号号扩展

无符号数扩展：简单填0补充。移位操作只能对无符号数进行，因为有符号的数右移可能导致符号位发生变化。当然视频音频可以例外。

原则：符号扩展只增加位数，不改变数值大小。

### 5.3数据截断

发生在长数据缩减为短数据时，不论大小端，都是截取最低字节给短数据类型变量，如果高字节不全为1或者0会发生截断错误。

### 5.4整型溢出

有符号的称为溢出，无符号的称为整型反转。

##### 例子1：32bit的int范围

32系统中,int型数表示的范围是多少？ -（2^31）-> (2^31 - 1)

帮助记忆解释：假设是一个int型的是4个bit。

则考虑符号正数最大是0111(2进制表示)，其实是（2^3 – 1为7）；

考虑符号负数其实是1111，其实是-7为啥会是-8呢。

考虑0这个数。其源码表示有2种

+0:0000,其对应得补码就是0000

-0:1000，符号位除外，取反+1，10000,1丢弃还是0000。

因此计算机将-0当做最大的负数，因为0有+0可以表示。也就是为什么负数的范围比正数多1.

##### 例子2:-1的补码

源码：1000…0001 符号位外取反 + 1 –> 0xffffffff,即补码为0xffffffff

##### 例子3:unsigned 0xffffffff 如果变成signed？

在内存中表示为1111…1111,如果定义（转换成了）符号数，则需要转为源码（由负数的补码求源码）才知道具体表示多少。可知为-1

##### 例子4:

char c = 128;

unsigned char uc = 128;

unsigned short us = 0;

us = c + uc;

us = c + (short)uc;

us = (unsigned char)c + uc;

us = c + (char)uc;

求各个表达式的us值？

1. us = c + uc,中首先发生整型提升，c和uc都会被提升int.则

c的补码从 1000000->1111…1000000

uc的补码从 1000000->0000…1000000,相加变为0000…0000截断为00..0000(16个)

1. us = c + (short)uc uc即便转成short也会提升为int，结果如上边，也是0
2. us = (unsigned char)c + uc，c由由有符号的强制转为无符号的，因此

c和uc的补码都为0000…1000,两个相加为0000…10000000,再截断为short,还是0000000100000000为0x100

1. us = c + (char)uc，uc强制转为有符号的，因此与c的补码相同，都为

1000000->1111…1000000，相加得到1111…111100000000.截断为0xff00

### 5.5 算数转换法则

#### (1)整型提升

首选在编译机来看，char、short、等都会被提升为int型来进行计算。这个就是整型替提升

#### (2)普适转换法则

当2个算术类型的操作数参与运算及逻辑运算时如a+b，如果2边类型不同，会进行自动转换，规则如下：

1. 如果一边为long double,则另一边也转为long double
2. 如果一边为double,则另一边也转为double
3. 如果一边float,则另一边也转为float
4. 如果2边同时为正数，首先做“整型提升”，如果类型仍然不同，继续替换。

规定：对于无符号和有符号的同一类型的整数，rank是相同的rank可简单认为字节数；

* 1. 2边符号相同的情况下，低rank向高rank转换

Rank Int > short > char

* 1. 1边有符号，1边无符号
     1. 如果无符号rank>=有符号的rank 转为 无符号的rank
     2. 无符号的rank < 有符号的rank，又分为如下2个情况
        1. 有符号的类型能覆盖无符号类型的数值，则转为有符号类型的
        2. 不能覆盖，则转为该个有符号类型数的对应的无符号类型

见例2

##### 例子1:

char c = 128;

unsigned char uc = 128;

unsigned short us = 0;

us = c + uc;

us = c + (short)uc;

us = (unsigned char)c + uc;

us = c + (char)uc;

求各个表达式的us值？

1. us = c + uc,中首先发生整型提升，c和uc都会被提升int.则

c的补码从 1000000->1111…1000000

uc的补码从 1000000->0000…1000000,相加变为0000…0000截断为00..0000(16个)

1. us = c + (short)uc uc即便转成short也会提升为int，结果如上边，也是0
2. us = (unsigned char)c + uc，c由由有符号的强制转为无符号的，因此

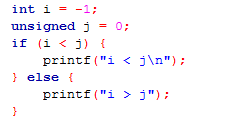
c和uc的补码都为0000…1000,两个相加为0000…10000000,再截断为short,还是0000000100000000为0x100

1. us = c + (char)uc，uc强制转为有符号的，因此与c的补码相同，都为

1000000->1111…1000000，相加得到1111…111100000000.截断为0xff00

##### 例子2:

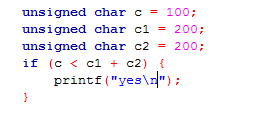
（1）如下代码输出什么？



I > j

分析：I < j进行运算时，有符号的int会转化为无符号的int(普适转换法则，无符号的rank>=有符号的rank),因此-1在软件中补码表示：0xffffffff,转为无符号的话，就是0xffffffff(正数的补码源码相同，因此是个很大的数)。

（2）如下代码输出什么？

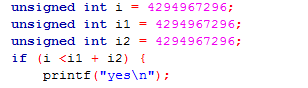


输出yes

分析：

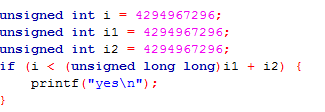
在计算c1+c2时，由于整型提升，被提升为int,因此unsigned char+unsigned char才不会溢出(反转)。在进行比较时，c也被提升为int型，但这个并不影响比较的结果。

（3）如下代码输出什么？



不会输出yes.

分析：i1+i2计算时，i1与i2已经是int,因此不会发生整型提升，因此2者相加发生了反转，导致I是比较大的。



## 6.预处理器与宏定义

（1）#：

把宏的参数直接替换为字符串，等价于直接把参数加上双引号。用在预编译语句中可以吧预编译函数的变量直接格式化为字符串。

（2）##：

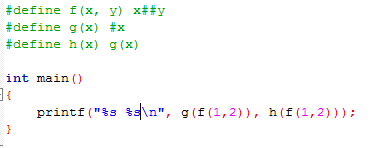
把2个参数粘在一起

注意：

当宏参数是另一个宏时，需要注意凡是当前的宏定义里有“#”或“##”的地方宏参数不会在展开。

### 例子1：

如下代码输出什么？



f(1,2) 12

分析:

由于g(f(1,2))属于 宏函数的参数为宏，并且g(x)的定义中有#,因此f(1,2)不会在这里展开，而是原封不动的打印。

而h(f(1,2)),也属于宏函数的参数是宏，但是h(x)的定义没有#(虽然g(x)中有也没有用)，其实这是转换宏，因此f(1,2)会在此展开，所以打印12

## 7.内联函数（inline）与宏定义

### （1）相同点：

1.内联函数的调用和宏函数有点类似，在调用点将代码展开，而不是开辟函数栈

### （2）不同点

1.编译器会对内联函数的参数类型做安全检查或自动类型转换，而宏定义不会。

2.内联函数可以调试。

### （3） 限制

1.内联函数只适合函数体内部代码简单的函数使用，不能包含结构控制语句例如while,switch，并且不能是递归函数。

2.inline函数仅仅是一个对编译器的建议，所以是否内联不一定的。

3.声明与定义要一致。Inline只有与函数体定义一起出现才有意义。如下的情况func是不会成为内联函数的

Inline void func(int x, int y);

Void func(int x, int y){}

## 8.typedef与宏定义

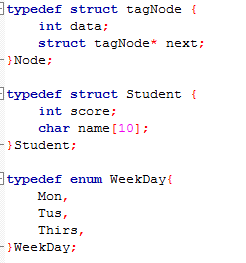
Typdef与define都是替一个对象取一个别名，以此增强程序的可读性，但是使用上和作用上都有如下区别。

1. 原理不同。#define是预编译指令，是语法，在预处理时只是简单的将字符串替换，不做正确性检查，只有在编译已被展开的源程序时才会发现可能的错误而报错。Typedef是关键字，在编译时处理，具有类型检查功能。
2. 功能不同。#define 不仅可以给类型取别名，还可以定义常量、变量、编译开关。

Typedef用来定义类型别名，包括内部类型及自定义类型。

1. 作用域。#define没有作用域，只要宏定义出现在使用之前都可以。

如下是typedef几个用法：



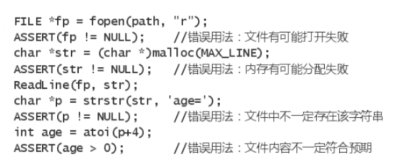
## 9.断言assert

断言是一种除错机制，用于验证代码是否符合编码人员的预期。编码人员在开发期间应该对函数的参数、代码中间执行结果合理地使用断言机制。断言被触发，说明程序出现了不应该出现的严重错误，会立即停止执行。

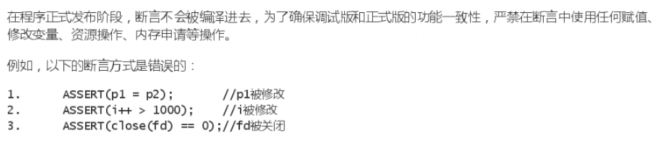
1. 断言必须使用宏定义，只在调试版本使用，最终版本不允许出现assert函数。



1. 运行时可能会导致出现的错误，严禁使用断言



1. 严禁在断言内改变运行环境



## 10.malloc与free

malloc函数返回的地址（指针）从变量存储的章节可知其指向的内存空间其实堆中的。

如int\* ptr = (int\*)malloc(10 \* sizeof(int))，其中ptr是属于栈上的变量，但其指向的地址是堆中分配的。

1. malloc与free必须配套使用，否则会出现内存泄漏。
2. malloc出的空间需要用memset进行初始化，否则容易出现意想不到的结果。当然可以使用calloc函数，与malloc一样也是在堆中申请开辟一段内存空间，但是还会将其默认初始化为0.

如上面代码等价于int \*ptr = (int\*)calloc(10, sizeof(int))

1. free之后的指针变量建议立即赋值，否则可能出现二次free（导致程序崩溃)或者被作为野指针被其他调用者调用。

## 11.指针

指针的定义在网上都能搜到。很难准确的说清，这里抓住主要矛盾，将难分清的说明一下，供大家理解。

### 11.1指针与数组

#### 相同点：

1.都表示一段内存的起点。

2.有相同的运算，如数组（指针）+ 偏移，下标引用[].

3.数组可以向指针赋值。Char str[10]; char\* p = str;

4.当数组作为函数的形参时，可以用指针做实参，因为void func(char str[])其实str是退化为指针的。

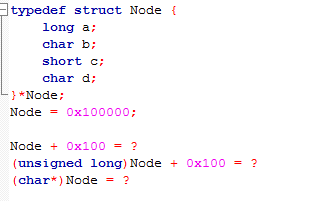
#### 不同点：

1.数组名是常量，指针是变量。因此任何对数组名进行自增运算是非法的。

2.指针只是说明在一个指针型内存单元中存放的是另一段内存中的值，改变后表示指向另一段新的内存的起始地址。

##### 例子1：

32位，4字节对齐时：



0x100C00 0x100100 0x100100

分析：

32bit下，4字节对齐Sizeof(Node)占12字节。

指针运算，其实是跳过指针指向数据的字节数。

因此Node + 0x100 其实是跳过0x100个sizeof(Node)

(unsigned long)Node 将Node的地址简单转化为数字，因此运算就是纯数字运算。

(char\*)Node将Node转化为char\*指针，因此运算，是跳过0x100个sizeof(char)

后2者的运算相同，但是含义不同。

延伸：

Int xs[2][2]

(&xs + 1) 其实跳过了2 \* 2 \* 4个字节，因为&xs是一个指向2\*2数组的指针。

### 11.2指针数组、数组指针、函数指针

三个元素，\*与[]的优先级是[] > \* ,因此[]会优先与p结合。

指针数组是数组，数组指针是指针。

#### （1）指针数组：

Int\* p[10] 表示p是一个拥有10个int\*的元素的数组，每个元素指向int型数据。

#### （2）数组指针

Int (\*p)[10] 表示p是指向一个有10个int型元素的数组的指针。

int\* (\*p)[10] 表示p是指向一个有10个int\*元素的数组的指针。

#### 函数指针

int (\*p)(void) p表示一个函数指针，返回值是int,函参无

int\* (\*p)(int\* a) p表示一个函数指针，返回值是int\*,函数参数为int\*

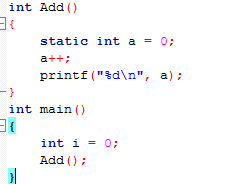
int\* (\*p)[10](int\* a)表示p是一个数组指针，数组的元素是函数指针，这种函数返回int\*，函数参数为int\*

## 12.static

### （1）对初始化影响

只影响局部变量。Static定义的局部变量指被初始化一次，下一次依据上一次的值。

#### 例子1：



以上代码打印1,2。不是1,1

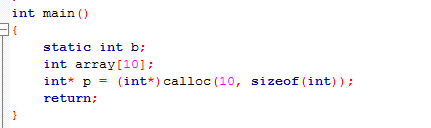
### 存储方式

全局变量本身就是存在BSS段或者data段，属于静态区；static修饰，仍然处于静态区，因此无本质变化。

局部变量本来存储在栈中，修饰后存储在静态区，因此发生了变化。

#### 例子1：

如下各个变量存储位置？



b:放在静态区，可能是BSS或者DATA

array:栈中

p:堆中

### 作用域

只影响全局变量，因为局部变量本省只对所在的函数可见。

用static修饰，导致该变量的作用域发生改变，变成只在当前文件中可见。

#### 例子1：

如下代码输出什么?

a.h

static int g\_count 0;

void Add(int x);

a.c

#include "a.h"

void Add(int x)

{

g\_count++;

}

b.c

#include "a.h"

void AddInt(void)

{

g\_count++;

}

int main()

{

printf("%d\n",g\_count);

AddInt();

printf("%d\n",g\_count);

Add(3);

printf("%d\n",g\_count);

}

0,1,1

分析:a.h声明的是static型的全局变量，因此包含了a.h的c文件都有自己可见的g\_count,y因此b.c中的main函数Add（3）其实最终的操作只影响a.c中的g\_count,并不影响b.c中的变量。

### 声明周期

影响全局和局部变量。

Static定义的变量生命周期贯穿于整个程序（main结束仍然存在，因为main也是被调用的）。

## 13.const

比较难区别的就是指向const的指针与const指针，其实就是看const后跟的是什么，跟的是变量名称就是修饰变量。

### （1）指针常量

const int\* p ;或者int const\* p ;

2种情况const都没有修饰p，因此是p指向的内容不能改变，假设最开始\*p = 5;那么\*p = 10 是非法的。

### （2）常量指针

int\* const p

const修饰p，可以知道p这个变量不能改了，其不能重定向到别的地址。Int b = 10;

p = &b是非法的

### （3）常量

Const int b 与int const b是等价的